Programming Languages
Lecture Notes

Nate Nystrom

7 March 2013

1 Operational semantics review

We can define the dynamic semantics of a language using a *small-step operational semantics*.

Consider a simple language with booleans and if. We can define the syntax of the language as terms t, some of which are values v.

$$t ::= v \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$$
$$v ::= \texttt{true} \mid \texttt{false}$$

Evaluation is defined using inference rules. We have two axioms for when the condition has already been evaluated:

$$\texttt{if true then}\ t_1\ \texttt{else}\ t_2 \longrightarrow t_1 \qquad \text{(E-TRUE)}$$

$$\texttt{if false then}\ t_1\ \texttt{else}\ t_2 \longrightarrow t_2 \qquad \text{(E-FALSE)}$$

Then, we have an inference rule that describes the order of evaluation.

$$\frac{t_0 \longrightarrow t_0'}{\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \longrightarrow \texttt{if}\ t_0'\ \texttt{then}\ t_1\ \texttt{else}\ t_2} \qquad \text{(E-IF)}$$

We can read the judgment $t \longrightarrow t'$ as "t reduces to t'". t (before the $\longrightarrow$) is called a *redex*. These rules are called *reduction rules* or *evaluation rules*. E-TRUE and E-FALSE are also called *computation rules*—they describe actual computation; E-IF is called a *congruence rule*—these rules describe order of evaluation..

Evaluation strategy. To evaluate a term t, we find a rule whose redex matches t and apply the rule, resulting in a new term t'. We *instantiate* the rule on t. If the rule has premises, those must also be matched.

Note the rule must match the entire term, so we cannot, for instance, apply a rule to an arbitrary subterm of an `if`. Thus

```
if true then (if false then false else false) else true
```

does not evaluate to

```
if true then false else true
```

The rules only let us evaluate the outer conditional first. The evaluation must step to

```
if false then false else false
```

first by E-TRUE, and then step to `false` by E-FALSE.

An evaluation step is justified by a derivation of the evaluation judgment using the inference rules. Let:

$$s = \textbf{if}\ \texttt{true}\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{false}$$
$$t = \textbf{if}\ s\ \textbf{then}\ \texttt{true}\ \textbf{else}\ \texttt{true}$$
$$u = \textbf{if}\ \texttt{false}\ \textbf{then}\ \texttt{true}\ \textbf{else}\ \texttt{true}$$

Then, we have the derivation tree (or proof tree):

$$\dfrac{\dfrac{\dfrac{}{s \longrightarrow \texttt{false}}\ \text{E-TRUE}}{t \longrightarrow u}\ \text{E-IF}}{\textbf{if}\ t\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{false} \longrightarrow \textbf{if}\ u\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{false}}\ \text{E-IF}$$

2 Induction

Here's a short review of proofs by induction. We'll first consider induction on natural numbers. This is probably what you're used to from, say, a data structures class.

Theorem 1. *For $n \geq 1$,*

$$1 \cdot 2 + 2 \cdot 3 + \cdots + n(n+1) = n(n+1)(n+2)/3$$

Proof. **Base case.** We first consider the base case $n = 1$. Clearly,

$$1 \cdot 2 = 1 \cdot 2 \cdot 3/3 = 1(1+1)(1+2)/3.$$

Induction case. Now, we have to show that if we assume the theorem is true for k, we can then show it is true for $k+1$. This assumption is called the *induction hypothesis* (IH).

In this proof, the IH is:

$$1 \cdot 2 + 2 \cdot 3 + \cdots + k(k+1) = k(k+1)(k+2)/3$$

We need to show:

$$1 \cdot 2 + 2 \cdot 3 + \cdots + k(k+1) + (k+1)(k+2) = (k+1)(k+2)(k+3)/3$$

By the IH, we have:

$$1 \cdot 2 + 2 \cdot 3 + \cdots + k(k+1) + (k+1)(k+2) = k(k+1)(k+2)/3 + (k+1)(k+2)$$

And so:

$$\begin{aligned} k(k+1)(k+2)/3 + (k+1)(k+2) &= (k+1)(k+2)k/3 + (k+1)(k+2) \\ &= (k+1)(k+2)\,(k/3+1) \\ &= (k+1)(k+2)\,(k/3+3/3) \\ &= (k+1)(k+2)\,((k+3)/3) \\ &= (k+1)(k+2)(k+3)/3 \end{aligned}$$

And this is what we wanted to prove.

□

Sometimes the obvious induction hypothesis is too weak and we need a stronger hypothesis. For instance, we might need to assume the property P we'd like to prove is true for all values $\leq k$, not just for the previous value k.

3 Structural induction

In general, you can apply the induction principle to anything where we have a partial order $\preceq$ where there are base objects for which no other object is strictly $\prec$.

For proving things about programs, we often use *structural induction*. This is based on the idea that there are simple terms (e.g., `true`, and `false`) and compound terms (e.g., **`if`**).

To prove a property about all possible terms, we prove it for the simple terms, and then assuming the property holds for all subterms of a compound terms, we can prove it for the compound term, then the property does indeed hold. Note that subterms of a term are "smaller" than the term itself, so we do have an ordering.

Here's an example:

Theorem 2. *Given the grammar and reduction rules above, either t is a value, or there is an t' such that $t \longrightarrow t'$.*

Proof. By structural induction on t.

- case $t =$ `true`. t is a value. Trivial.
- case $t =$ `false`. t is a value. Trivial.
- case $t =$ **`if`** t_0 **`then`** t_1 **`else`** t_2. Since t_0 is a subterm of t, by the IH, t_0 is either a value or there is a t_0' such that $t_0 \longrightarrow t_0'$.
  - subcase $t_0 =$ `true`. Then E-TRUE applies and $t' = t_1$.
  - subcase $t_0 =$ `false`. Then E-FALSE applies and $t' = t_2$.
  - otherwise, by the IH, $t_0 \longrightarrow t_0'$, so we can derive $t' =$ **`if`** t_0' **`then`** t_1 **`else`** t_2 by E-IF.

□

4 IMP

So far, we haven't talked about imperative languages, but we can of course model these too with operational semantics.

Here is the grammar for a language called IMP. Note that the language is restricted syntactically to ensure that variables contain only integers and that arithmetic and boolean expressions are distinct. This is to avoid the need to handle corner cases like `true`+1.

$s ::=$ **`pass`** $\mid x := a \mid s_1;\ s_2 \mid$ **`if`** b **`then`** s_1 **`else`** $s_2 \mid$ **`while`** b **`do`** s	statements
$a ::= x \mid n \mid a_1 + a_2$	arithmetic expressions
$b ::=$ **`true`** $\mid$ **`false`** $\mid a_1 < a_2 \mid b_1$ **`and`** $b_2 \mid$ **`not`** b	boolean expressions

Here are the operational semantics. Statement evaluation is defined by judgments of the form $\sigma, s \longrightarrow \sigma', s'$ ("s in store σ reduces to s' in σ'"). Evaluation halts in the configuration $\sigma,$ **`pass`**. A store σ is a function from variables x to values n. $\sigma[x \mapsto n]$ is the store that maps x to n and y ($\neq x$) to $\sigma(y)$.

$$\sigma, \textbf{pass};\ s \longrightarrow \sigma, s \qquad \text{(S-SEQ)}$$

$$\frac{\sigma, s_1 \longrightarrow \sigma', s_1'}{\sigma, s_1;\ s_2 \longrightarrow \sigma', s_1';\ s_2} \qquad \text{(SC-SEQ)}$$

$$\sigma, x := n \longrightarrow \sigma[x \mapsto n], \textbf{pass} \qquad \text{(S-ASN)}$$

$$\frac{\sigma \vdash a \longrightarrow a'}{\sigma, x := a \longrightarrow \sigma, x := a'} \quad \text{(SC-ASN)}$$

$$\sigma, \texttt{if true then } s_1 \texttt{ else } s_2 \longrightarrow \sigma, s_1 \quad \text{(S-IFTRUE)}$$

$$\sigma, \texttt{if false then } s_1 \texttt{ else } s_2 \longrightarrow \sigma, s_2 \quad \text{(S-IFFALSE)}$$

$$\frac{\sigma \vdash b \longrightarrow b'}{\sigma, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \longrightarrow \sigma, \texttt{if } b' \texttt{ then } s_1 \texttt{ else } s_2} \quad \text{(SC-IF)}$$

$$\sigma, \texttt{while } b \texttt{ do } s \longrightarrow \sigma, \texttt{if } b \texttt{ then } (s;\ \texttt{while } b \texttt{ do } s) \texttt{ else pass} \quad \text{(S-WHILE)}$$

Note that the rule for **while** is non-obvious. We copy the original **while** loop into the target statement. This makes it possible to write infinite loops:

$$\begin{aligned}\sigma, \texttt{while true do pass} &\longrightarrow \sigma, \texttt{if true then } (\texttt{pass};\ \texttt{while true do pass}) \texttt{ else pass} \\ &\longrightarrow \sigma, \texttt{pass};\ \texttt{while true do pass} \\ &\longrightarrow \sigma, \texttt{while true do pass} \\ &\longrightarrow \ldots\end{aligned}$$

Expression evaluation is defined by judgments of the form $\sigma \vdash e \longrightarrow e'$ ("e reduces to e' with store σ"). Note that we're not stepping to a new store σ' in these rules because expressions do not include assignment. You can think of $\sigma \vdash e \longrightarrow e'$ as a rewrite rule for expressions e parameterized on a store σ.

To simplify the rules, we add the following syntax:

$e ::= a \mid b$	expressions
$v ::= n \mid \texttt{true} \mid \texttt{false}$	values
$o ::= + \mid < \mid \texttt{and}$	binary operations

$$\sigma \vdash x \longrightarrow \sigma(x) \quad \text{(E-VAR)}$$

$$\sigma \vdash n_1 < n_2 \longrightarrow \texttt{true} \qquad (\text{where } n_1 < n_2) \quad \text{(E-LT)}$$

$$\sigma \vdash n_1 < n_2 \longrightarrow \texttt{false} \qquad (\text{where } n_1 \geq n_2) \quad \text{(E-GE)}$$

$$\sigma \vdash n_1 + n_2 \longrightarrow n \qquad (\text{where } n = n_1 + n_2) \quad \text{(E-ADD)}$$

$$\sigma \vdash \texttt{false and } b \longrightarrow \texttt{false} \quad \text{(E-ANDFALSE)}$$

$$\sigma \vdash \texttt{true and } b \longrightarrow b \quad \text{(E-ANDTRUE)}$$

$$\frac{\sigma \vdash e_1 \longrightarrow e_1'}{\sigma \vdash e_1 \ o \ e_2 \longrightarrow e_1' \ o \ e_2} \quad \text{(EC-L)}$$

$$\frac{\sigma \vdash e \longrightarrow e'}{\sigma \vdash v \ o \ e \longrightarrow v \ o \ e'} \quad \text{(EC-R)}$$

$$\sigma \vdash \texttt{not true} \longrightarrow \texttt{false} \quad \text{(E-NOTTRUE)}$$

$$\sigma \vdash \texttt{not false} \longrightarrow \texttt{true} \quad \text{(E-NOTFALSE)}$$

$$\frac{\sigma \vdash e \longrightarrow e'}{\sigma \vdash \texttt{not } e \longrightarrow \texttt{not } e'} \quad \text{(EC-NOT)}$$

Let's do an example. Let σ be an initial store. Here are the derivations for each step of a computation.

$$\dfrac{\dfrac{}{\sigma, x := 1 \longrightarrow \sigma[x \mapsto 1], \textbf{pass}}\ \text{S-Asn}}{\sigma, x := 1; y := x + 1 \longrightarrow \sigma[x \mapsto 1], \textbf{pass}; y := x + 1}\ \text{SC-Seq}$$

$$\dfrac{}{\sigma[x \mapsto 1], \textbf{pass}; y := x + 1 \longrightarrow \sigma[x \mapsto 1], y := x + 1}\ \text{S-Seq}$$

$$\dfrac{\dfrac{\dfrac{}{\sigma[x \mapsto 1] \vdash x \longrightarrow 1}\ \text{E-Var}}{\sigma[x \mapsto 1] \vdash x + 1 \longrightarrow 1 + 1}\ \text{EC-L}}{\sigma[x \mapsto 1], y := x + 1 \longrightarrow \sigma[x \mapsto 1], y := 1 + 1}\ \text{SC-Asn}$$

$$\dfrac{\dfrac{}{\sigma[x \mapsto 1] \vdash 1 + 1 \longrightarrow 2}\ \text{E-Op}}{\sigma[x \mapsto 1], y := 1 + 1 \longrightarrow \sigma[x \mapsto 1], y := 2}\ \text{SC-Asn}$$

$$\dfrac{}{\sigma[x \mapsto 1], y := 2 \longrightarrow \sigma[x \mapsto 1, y \mapsto 2], \textbf{pass}}\ \text{S-Asn}$$

5 Determinism

Now, let's prove that IMP programs are determinsitic.

To do the proof, we need the following lemma, the proof of which is an exercise in HW 3.

Lemma 1. *If $\sigma \vdash e \longrightarrow e'$ and $\sigma \vdash e \longrightarrow e''$, then $e' = e''$.*

The main theorem follows:

Theorem 3. *If $\sigma, s \longrightarrow \sigma, s'$ and $\sigma, s \longrightarrow \sigma, s''$, then $s' = s''$.*

Proof. By structural induction on s.

- case $s = \textbf{pass}$. Vacuous (s' and s'' do not exist).
- case $s = x := a$.
  There are two cases. If a is n, then $s' = \textbf{pass} = s''$ by S-Asn.
  Otherwise, by the lemma $\sigma \vdash a \longrightarrow a'$ uniquely and we can derive $s' = x := a' = s''$ uniquely by SC-Asn.
- case $s = s_1;\ s_2$.
  If $s_1 = \textbf{pass}$, then $s' = s'' = s_2$ by S-Seq.
  Otherwise, by the IH, If $\sigma, s_1 \longrightarrow \sigma', s_1'$ uniquely. By SC-Seq, we can derive a unique $s' = s'' = s_1'; s_2$.
- case $s = \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2$
  There are three cases. If b is **true** or **false**, then by S-IfTrue, we have $s' = s'' = s_1$ or by S-IfFalse, we have $s' = s'' = s_2$.
  Otherwise, by the lemma b' is unique. Therefore by SC-If, $s' = s'' = \textbf{if}\ b'\ \textbf{then}\ s_1\ \textbf{else}\ s_2$.
- case $s = \textbf{while}\ b\ \textbf{do}\ s_1$
  Only one rule applies. Trivial.

□